Microsoft
.net developer
training
Create XML Web Services and Applications with Visual Studio .NET

Application Interoperability

Agenda: Day 2

Module 09: Creating XML Web Services

Module 10: Code Instrumentation and Middle Tier Technology

Module 11: Security in .NET

Module 12: Lifecycle Tools for .NET

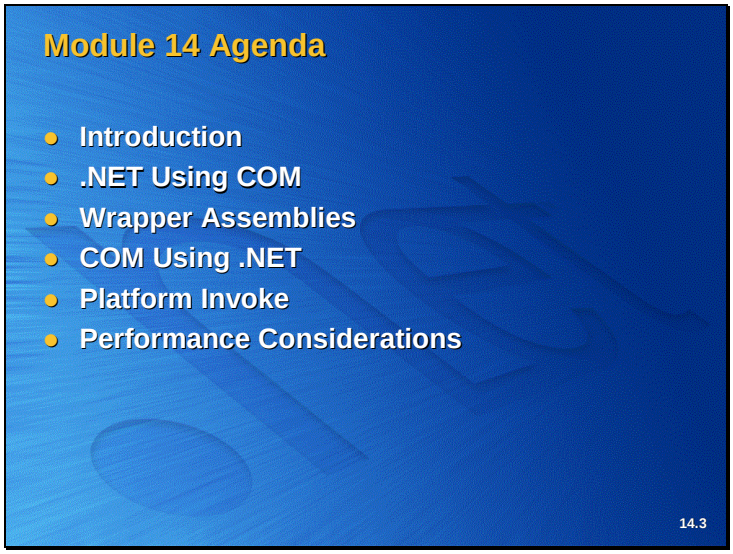
Module 13: Configuring and Deploying .NET Applications

▶ Module 14: Application Interoperability

Module 15: Migrating Applications to .NET

Module 16: Building Applications for Mobile Devices

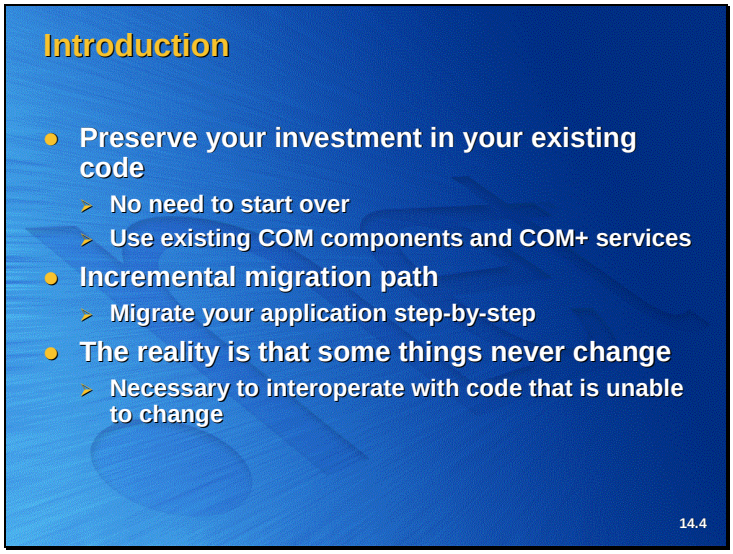
 ©2002 Microsoft Corporation



Module 14 Agenda

What you will learn:

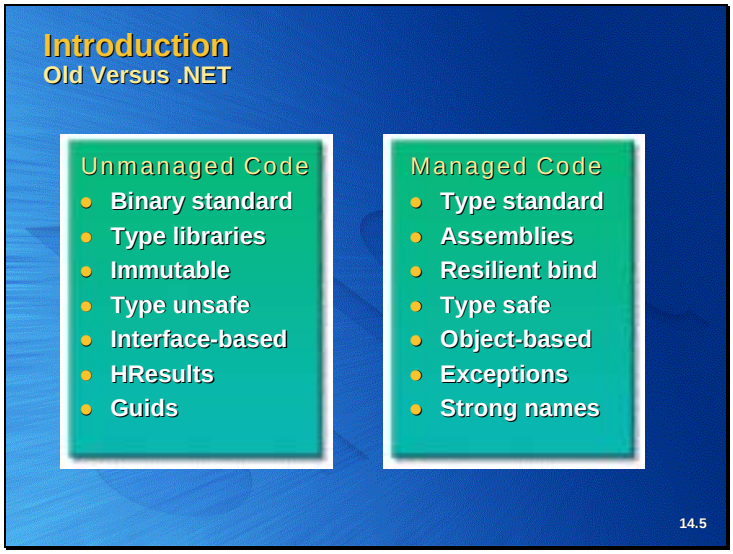
- How .NET works with COM and COM+
- What COM Interop is and what it does for you


Introduction
Preserve your investment in your existing code
No need to start over
Use existing COM components and COM+ services
Incremental migration path
Migrate your application step-by-step
The reality is that some things never change
Necessary to interoperate with code that is unable to change
14.4

Introduction

The **Microsoft® .NET Framework** promotes interaction with COM components, COM+ services, external type libraries and many operating system services. Data types, method signatures and error-handling mechanisms vary between managed and unmanaged Object models. To simplify interoperation between .NET Framework components and unmanaged code and to ease the migration path, the Common Language Runtime hides the differences in these Object models from both clients and servers.

 ©2002 Microsoft Corporation



Introduction

Old versus .NET

Type definitions
If you are accustomed to working with **type libraries**, you know that they contain public types only. Moreover, a type library is optional unless in the managed programming model, where type information is mandatory. Interoperation services provide tools that convert type libraries to metadata in assemblies and metadata to type libraries.

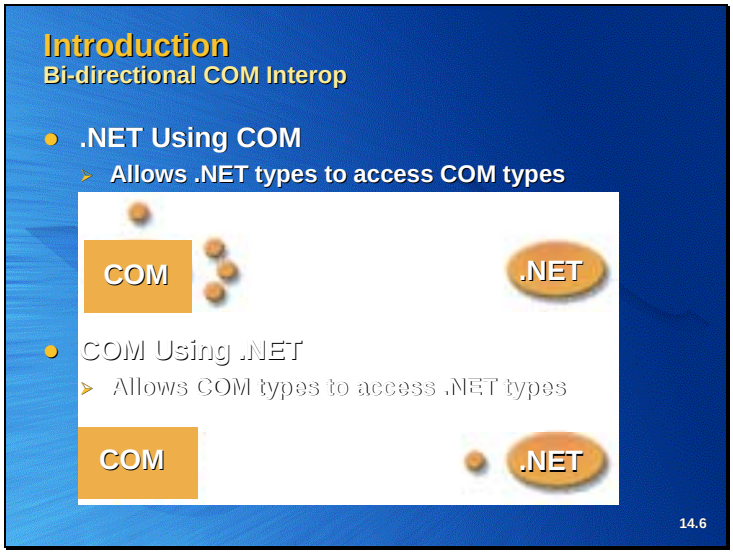
Types vary between managed and unmanaged code, but also among languages.

Versioning
COM interfaces are immutable. If you change an interface, you must rename it with a new GUID. Managed types can evolve while maintaining the same name.

Type safety
Unmanaged compilers provide no type checking on pointer types, making the code susceptible to malicious activity. In general, managed code requires a higher level of trust. Programmers can continue to use pointers in managed code, although the code has restrictions because of its unsafe behavior. Interoperation services prevent non-trusted, managed code from accessing unmanaged code.

Error-handling mechanisms
COM methods usually return an **HRESULT**, indicating that the call succeeded or failed. Managed code incorporates exceptions. By default, COM Interop maps managed exceptions to failure **HRESULTs**.

GUIDs
Identify a specific unmanaged type and provide no location information on that type. Strong names consist of a unique assembly name in addition to a type name. Because the assembly name uniquely identifies the type, you can reuse a type name across multiple assemblies. An assembly also introduces publisher key, version, and location information to a managed type. Interoperation services generate GUIDs and are strong-named as required.



Introduction

Bi-directional COM Interop

If you decide not to upgrade, Microsoft enables you to use your existing code through **COM Interop**.

One of the primary goals of the .NET Framework is to make COM development easier because one of the hardest things about COM development is dealing with the COM infrastructure. Consequently, to make COM development easier, the .NET Framework automates virtually all of what developers currently think of as COM, including reference-counting, interface description and registration.

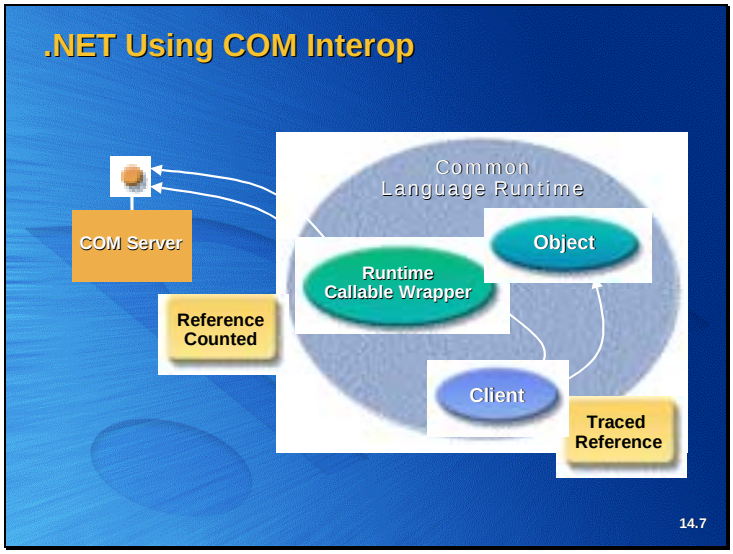
It is important to note that this doesn't mean that .NET Framework components aren't COM components. In fact, a COM developer using Visual Studio 6® could call a .NET Framework component and, to the developer, it would look like a COM component, complete with unknown data. Conversely, a .NET Framework developer using Visual Studio® .NET would see a COM component as a .NET Framework component.

There is a caveat to this relationship. COM developers must manually do many of the things that .NET Framework developers can rely on the CLR to automate for them. For example, the security of a COM component must be written manually, and its memory can't be automatically managed, and, to install a COM component, entries must be placed in the Windows registry. For .NET Framework components, the CLR automates these features. Components are self-describing, for example, and can therefore be installed without registering them in the Windows registry.

The CLR provides a common type system where applications can be written in multiple languages using closer integration than before. For example, a class written in Visual C#™ can be inherited by a class written in Visual Basic.

A set of services is provided in the form of class libraries. These provide the developer with encapsulated functionality such as Web services and SOAP along with the services provided by Windows and COM+. Interop is a stopgap. It isn't used as a replacement for creating a new managed Object model. COM Interop fails to provide many of the benefits of .NET. There are no parameterized constructs, inheritance, deployment, versioning support, zero-impact deployment, etc. It simply lets you get access to the stuff you have today.

 ©2002 Microsoft Corporation

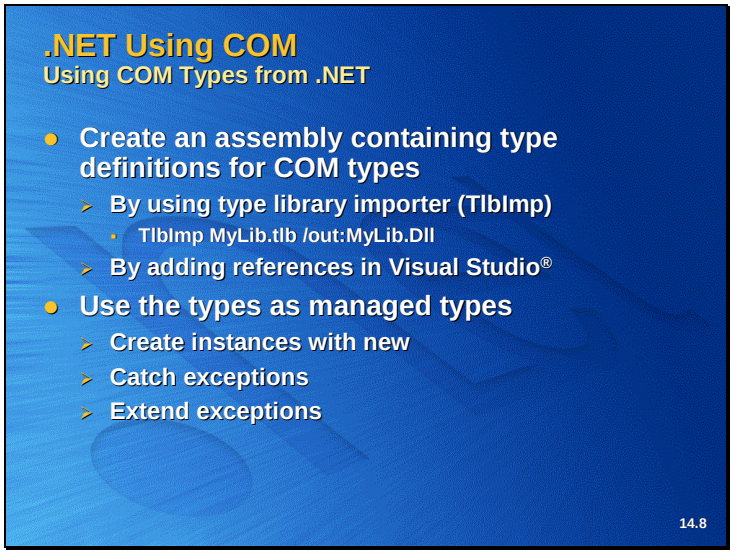


.NET Using COM Interop

A .NET Client accesses a COM server through a Runtime Callable Wrapper (RCW). The RCW is responsible for handling all interaction between the .NET Client code and the COM component, including (but not limited to):

- Creating and binding to the underlying COM object,
- Consuming COM interfaces and factoring the interfaces into a managed form.
- Translating and marshaling data between environments.
- Managing the lifetime of the wrapped COM object.
- Translating COM HRESULT values into .NET exceptions.

The RCW is a managed object and is allocated from the heap maintained by the CLR. As with any other managed Object, references to the RCW are traced by the CLR and the RCW is subject to garbage collection.


.NET Using COM
Using COM Types from .NET
Create an assembly containing type definitions for COM types
By using type library importer (TlbImp)
TlbImp MyLib.tlb /out:MyLib.Dll
By adding references in Visual Studio®
Use the types as managed types
Create instances with new
Catch exceptions
Extend exceptions
14.8

.NET Using COM

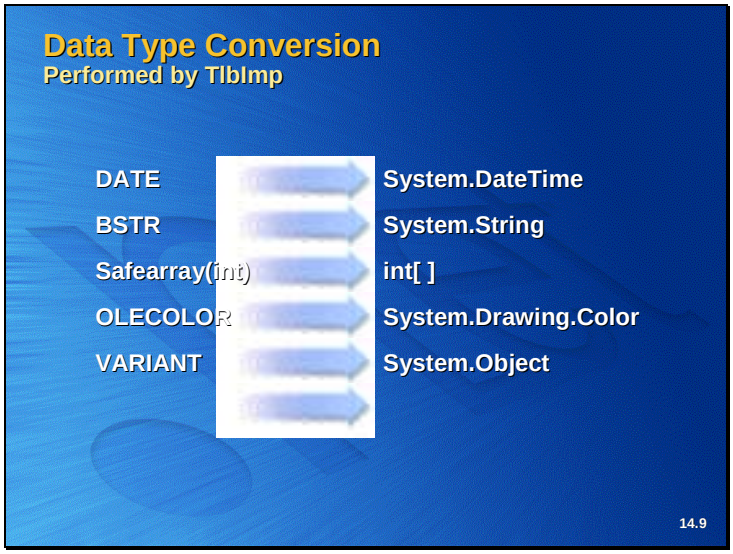
Using COM Types from .NET

The runtime requires metadata for all types, including COM types, in most situations. The **TlbImp** utility included in the SDK can convert COM type libraries to .NET Framework metadata. Once the type library is converted to metadata, managed clients can call the COM type seamlessly. For ease of use, always provide type information in a type library.

Don't think of managed types as COM types because there is:

- No reference counting
- No IDispatch
- No HResults
- No Connection Points
- No Guids

 ©2002 Microsoft Corporation



Data Type Conversion

Performed by TlbImp

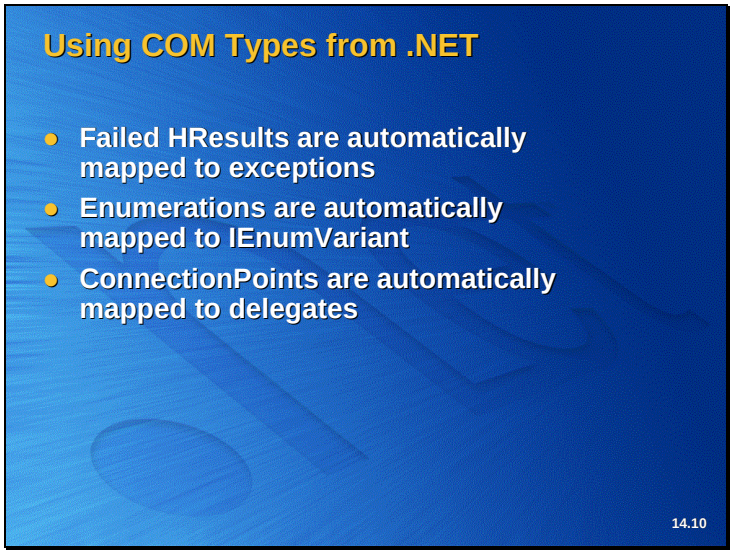
The Type Library Importer (TlbImp) is a command-line tool that converts the co-classes and interfaces contained in a COM type library to metadata. This tool creates an assembly and namespace for the type information automatically. After the metadata of a class is available, managed clients can create instances of the COM type and call its methods, just as if it were a .NET instance. The Type Library Importer tool converts an entire type library to metadata at once and cannot generate type information for a subset of the types defined in a type library.

See Appendix A for details on TlbImp and other .NET Utilities.

To generate an assembly from a type library
Use the following command to produce the Loanlib.dll assembly from the type library contained in Loanlib.dll (or to change extension to be clearer like Loanlib.tlb)

Adding the **/out:** switch produces an assembly with an altered name, such as LOANLib.dll. Altering the interop assembly name can help distinguish it from the original COM DLL.

```
c:\>tlbimp LoanLib.dll /out: LoanLibWrapper.dll
```


Using COM Types from .NET
Failed HResults are automatically mapped to exceptions
Enumerations are automatically mapped to IEnumVariant
ConnectionPoints are automatically mapped to delegates
14.10

Using COM Types from .NET

- When a managed client calls a COM Object, the runtime maps the COM Object's failure HRESULTs to exceptions, which the marshaler throws on return from the call.
- Enumerations mapped to IEnumVariant
- ConnectionPoints mapped to delegates

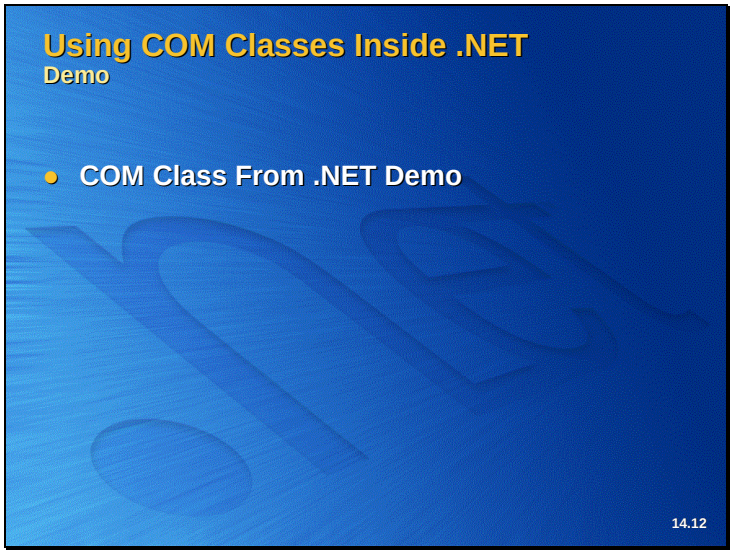
 ©2002 Microsoft Corporation

Installing Wrapper Assemblies
Assembly with COM type definitions must be resolvable at runtime
Installed in the application directory
Copy assembly along with the application file
Installed in the Global Assembly Cache
Generate key file in Visual Studio® .NET
Sign the assembly using TlbImp
C:\>TlbImp MyLib.tlb /keyfile:MyKey.snk
14.11

Installing Wrapper Assemblies

The .NET Framework SDK provides tools to import and convert a type library to an Interop assembly. You must ship the Interop assembly with the other assemblies that make up your application.

Assemblies can have strong names. A strong-named assembly includes the publisher's public key, which provides a unique identity. Assemblies produced by Type Library Importer (**Tlbimp.exe**) can be **signed by** the publisher using the **/keyfile** option. You can install signed assemblies into the GAC. Unsigned assemblies must be installed on the user's machine as private assemblies.



Using COM Classes Inside .NET

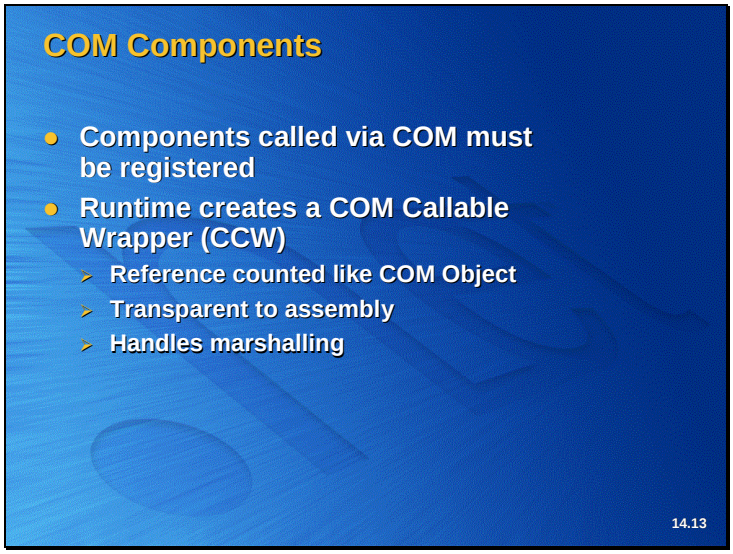
Demo

Show how to reference a COM dll and create the COM Interop wrapper.

The purpose of this demo is to:

- Show how to add references to a COM component
- Show how to add COM Interop wrapper with VS .NET
- Show how to use COM from .NET

 ©2002 Microsoft Corporation

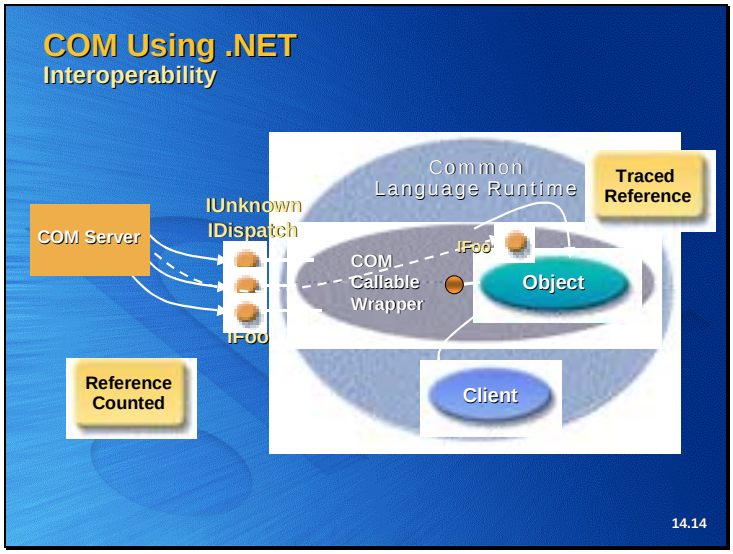


COM Components

The **COM Callable Wrapper** (CCW) wraps the .NET Object and mediates between it and the CLR environment, making the .NET Object appear to COM clients as if it were a native .NET Object.

If assembly will be in the same dir as COM client there is no need to sign it. It must be signed only if it'll be installed in GAC. It must follow standard .NET component location conventions, which means it must reside in the GAC, or less commonly, in the client application's directory tree. There is one more possibility, if .NET component is registered using /codebase switch than it can be in this dir also. Any .NET class that you want COM Using create must provide a default constructor, meaning a constructor that requires no parameters.

CLR doesn't need type libraries. COM does. To marshal calls correctly between incompatible apartments, the COM might need to locate the type library that describes a particular type. A type library must be registered before the COM can see it, except in the case of late binding. We can use regasm /tlb switch to generate the type library for .NET component and to register this type library in the same time.



COM Using .NET

Interoperability

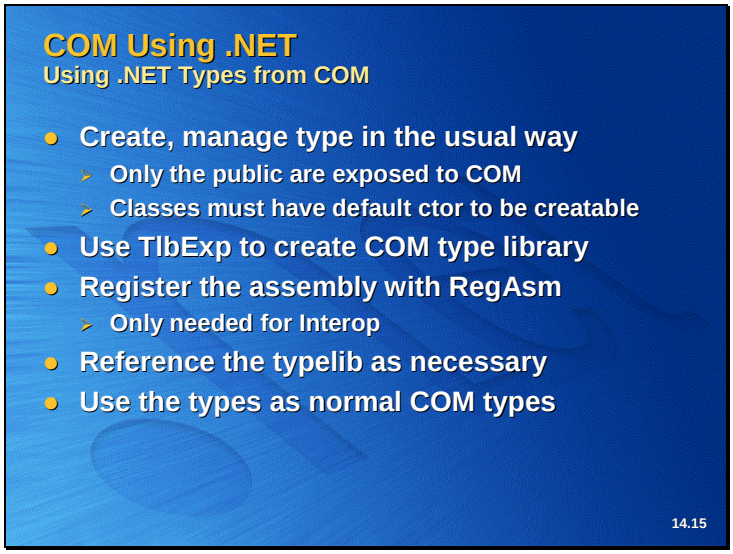
The **.NET Framework** supports this situation as well via a CCW.

The **CCW** is responsible for handling all interaction between the COM client and the managed Object, including (but not limited to):

- Creating and binding to the underlying managed Object
- Synthesizing several important COM interfaces (IUnknown, IDispatch and several other interfaces) based on the Object's type information
- Marshaling and translating data between environments
- Managing the lifetime of the .NET component
- Translating .NET exceptions into COM HRESULT values

The CCW is an unmanaged COM Object. It is allocated from the standard Windows heap and is reference counted similar to a traditional COM Object. It is not garbage collected, but is destroyed upon release of the last client reference. When the CCW is destroyed, the managed Object that it wraps is marked for garbage collection.

 ©2002 Microsoft Corporation


COM Using .NET
Using .NET Types from COM
Create, manage type in the usual way
Only the public are exposed to COM
Classes must have default ctor to be creatable
Use TlbExp to create COM type library
Register the assembly with RegAsm
Only needed for Interop
Reference the typelib as necessary
Use the types as normal COM types
14.15

COM Using .NET

Using .NET Types from COM

Guids are generated automatically at export time:

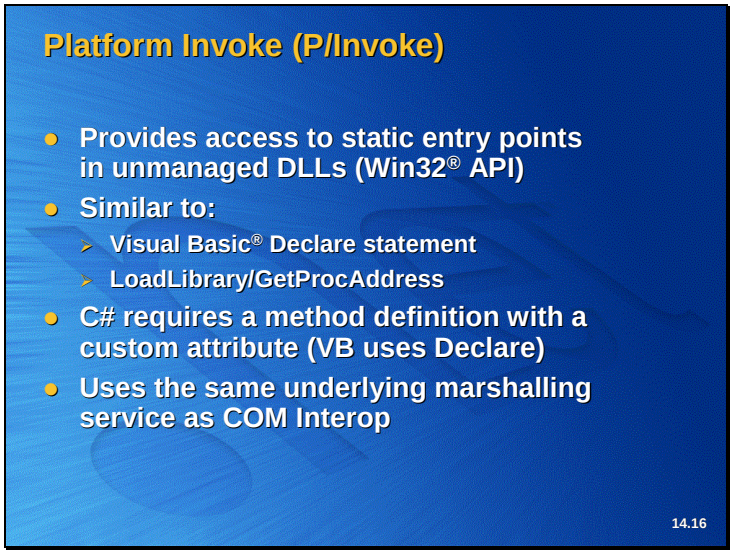
- Based on assembly and type name
- Based on complete interface definition

Typelib file may be needed at run time:

- Runtime will produce one automatically
- Pre-package along with assembly

Assembly must be resolvable at run time to activate from COM:

- Installed in application directory or installed in Global Assembly Cache


Platform Invoke (P/Invoke)
Provides access to static entry points in unmanaged DLLs (Win32® API)
Similar to:
Visual Basic® Declare statement
LoadLibrary/GetProcAddress
C# requires a method definition with a custom attribute (VB uses Declare)
Uses the same underlying marshalling service as COM Interop
14.16

Platform Invoke (P/Invoke)

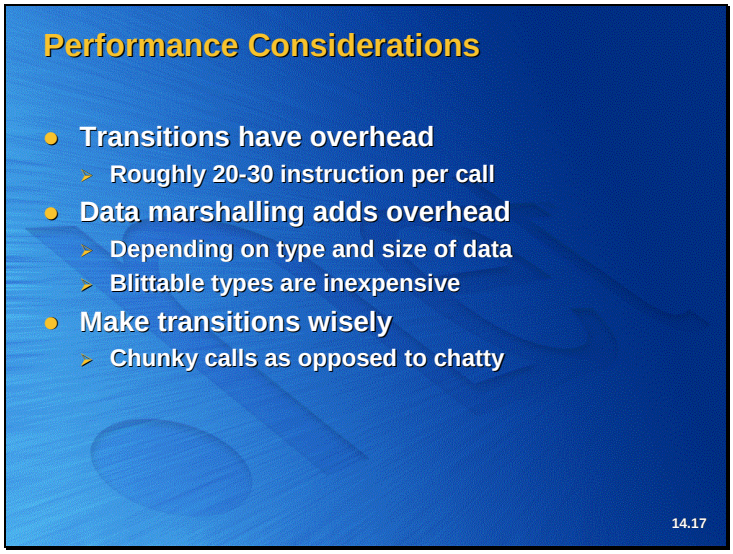
Example #1

```
public class Win32API {
    [DllImport("User32.dll", EntryPoint="MessageBox")]
  public static extern int MsgBox(int hWnd, String text, String
caption, uint type);
}
```

Example #2

```
[StructLayout(LayoutKind.Sequential)]
public int OSVersionInfoSize;
public int majorVersion;
public int minorVersion;
public int buildNumber;
public int platformId;
[ MarshalAs( UnmanagedType.ByValTStr, SizeConst=128 )]
public String versionString;
}
public class Win32API {
[DllImport("kernel32.dll")]
public static extern  bool GetVersionEx(ref OSInfo osi);
}
```

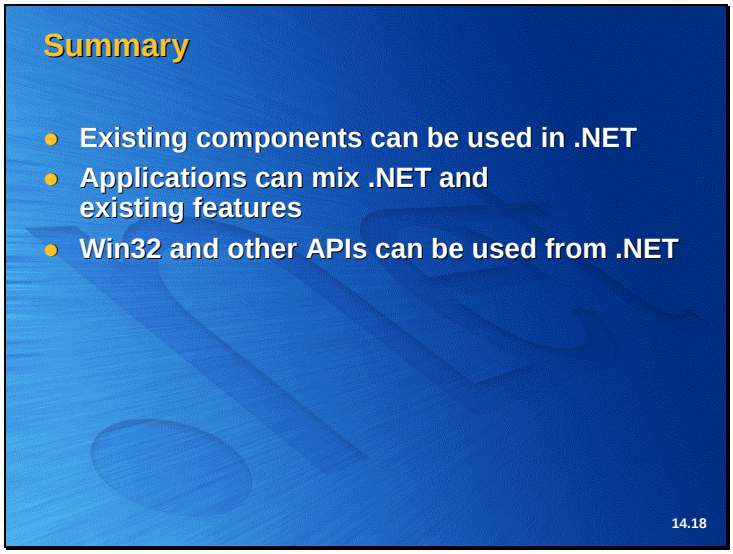
 ©2002 Microsoft Corporation



Performance Considerations

Marshalling data between managed and unmanaged code has a cost, but you can mitigate that cost by making fewer transitions across the boundary. Chunky interfaces that minimize the number of transitions generally perform better than **chatty interfaces** that cross the boundary often, performing small tasks with each crossing.

Blittable is the new term for isomorphic. Blittable types are the data types that have the same representation in managed as well as unmanaged memory. No conversion or marshalling is required when these are passed between managed and unmanaged code.


Summary
Existing components can be used in .NET
Applications can mix .NET and existing features
Win32 and other APIs can be used from .NET
14.18

Summary

You can use your existing components in .NET and you can use other APIs. COM Interop is an interim solution that lets you access your existing code using .NET components. COM Interop fails to provide many of the benefits of .NET. There are no parameterized constructs, inheritance, deployment, versioning support, zero-impact deployment, etc. It simply lets you get access to the stuff you have today.

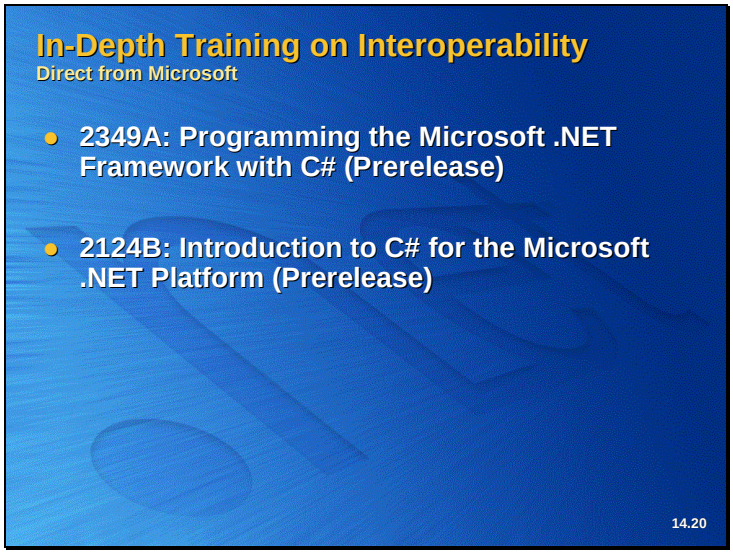
 ©2002 Microsoft Corporation

Resources
.NET Interop: Get Ready for Microsoft .NET by Using Wrappers to Interact with COM-based Applications by David S. Platt:
http://msdn.microsoft.com/msdnmag/issues/01/08/interop/interop.asp
Handling Language Interoperability with the Microsoft .NET Framework - This article looks at the Common Language Runtime and the Common Language Specification as well as metadata
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/interopdotnet.asp
14.19

Resources

.NET Interop: Get Ready for Microsoft .NET by Using Wrappers to Interact with COM-based Applications by David S. Platt:
http://msdn.microsoft.com/msdnmag/issues/01/08/interop/interop.asp

Handling Language Interoperability with the Microsoft .NET Framework - This article looks at the Common Language Runtime and the Common Language Specification as well as metadata:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/interopdotnet.asp


In-Depth Training on Interoperability
Direct from Microsoft
2349A: Programming the Microsoft .NET Framework with C# (Prerelease)
2124B: Introduction to C# for the Microsoft .NET Platform (Prerelease)
14.20

In-Depth Training on Interoperability

Direct from Microsoft

 ©2002 Microsoft Corporation

Microsoft
14.21